Dynamic Processor Modes in Puma*

Arthur B. Maccabe
maccabe@cs.unm.edu
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131

Rolf Riesen
rolf@cs.sandia.gov
Sandia National Laboratories
Organization 9223
Albuquerque, NM 87185-1110

David W. van Dresser
dwvandr@cs.sandia.gov
Sandia National Laboratories
Organization 9223
Albuquerque, NM 87185-1110

July 23, 1996

Abstract

In this paper, we describe the dynamic processor modes supported by Puma, an operating system for massively parallel computing systems. Puma was designed for computing systems that consist of thousands of nodes connected by a communication network with high bandwidth and low message-passing latency. The individual nodes of such a system may have multiple processors with a shared memory. This presents the potential to exploit parallelism at two levels: parallelism between nodes and parallelism within a node.

In this paper, we describe the mechanisms we have developed to support library and application writers in their efforts to exploit the parallelism provided by a single node. Following the Puma philosophy, the mechanisms provided by the kernel represent a very thin (but safe) level above the actual hardware. In addition to the kernel mechanisms, we also describe higher-level library functions that application writers or people writing runtime systems can use.

1 Background

The Puma [11, 13] operating system was designed to support high performance computing on massively parallel computing systems. We are primarily interested in supporting applications that consume a large portion of the resources offered by such a computing system (e.g., memory, processor cycles, communication bandwidth, etc.) for several hours or, perhaps, days. Examples of such applications include climate modeling and simulations of nuclear explosions.

High performance computing involves the management of at least one critical resource. For some applications the critical resource may be memory, for others it may be processor cycles, for still others the critical resource may be communication bandwidth. To support these applications, the Puma kernel provides a very thin, but safe, interface to the underlying hardware. In this respect Puma is similar to the MIT Exo kernel [5], which also attempts to push as much functionality as is safely possible up to the user level. Puma is not taking this idea to the same extremes as the Exo

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

kernel design does. Performance critical functions, such as the handling of incoming messages and the mechanisms that enforce the policies made at user level, all reside within the Puma kernel.

A high-level interface usually simplifies application development by providing a view that better matches the computational model used in the application. However, the implementation of a high-level interface, by definition, provides management for one or more resources. As an example, implementations of high-level languages provide management for registers, memory layout, and other resources provided by the machine. If the implementation of an interface manages a critical resource, this management may conflict with the application's management needs. When such a conflict arises, the application may not be able to meet its performance goals. The thin interface provided by the Puma kernel should minimize the impact of these conflicts.

Because the Puma kernel only provides a low-level interface, this interface may be obscure and cumbersome for many application programmers. Libraries provide interfaces with semantics that better match the models used by application programmers. Because these higher-level interfaces are in the libraries, they can be used when appropriate and easily circumvented when performance considerations mandate more direct management of the resources.

Current generation massively parallel computing systems have thousands of nodes connected by a high-performance communication network. The communication networks have bandwidths on the order of hundreds of megabytes per second with message passing latencies on the order of microseconds. While there may be some hardware support for distributed shared memory, communication between nodes is essentially based on explicit message passing.

In Puma, we manage the nodes in a massively parallel system using "space sharing." Users allocate a collection of nodes for their applications at load time. Puma supports multitasking on the nodes; however, the tasks running on a node are all running on behalf of a single user (and presumably on behalf of a single application). This approach lets the application programmer manage a known set of resources on each node. If we were to combine tasks from different users on a single node, as is done in traditional multitasking operating systems, the resources provided by the node to a given task would be dependent on the current mix of tasks. This situation would make it virtually impossible for the application programmer to manage the resources as needed.

Each node in a massively parallel computing system may have a small collection of processors (two to four) with a shared memory [12]. This organization provides two levels of parallelism to be managed: parallelism among the processors in a node, and parallelism among the nodes within the system.

In the Puma programming model an application consists of a collection of tasks that communicate using explicit message passing. Each task runs on a single node of a massively parallel system. When a task is distributed among the processors on a node, the portion of the task that runs on a processor might naturally be called a "thread." We have intentionally avoided this terminology because of the semantic baggage that is typically associated with the term thread.

In Puma, the processors on a node are *gang scheduled*: when a task is scheduled for execution, all of the processors on the node are available to the scheduled task. We do not attempt to run different tasks (from the same application) on the individual processors of a node. When an application task is running on a node, all of the processors share a common address space (page table).

Before we discuss the dynamic models of processor usage supported by Puma, we consider *virtual nodes*. When there are multiple processors on a node, one possible use of the processors is to have each processor emulate a virtual node. Puma supports virtual nodes. For the most part, virtual nodes act just like physical nodes. One difference is that the text segment of a single physical node can be shared (read-only) by all CPUs on that node—Puma does not currently support any other form of sharing across different address spaces. Unlike the other multi-processor usage models that we describe in this paper, using the processors as virtual nodes is a *static* decision. The user must select virtual nodes when the node is allocated and this selection remains in effect until the node is de-allocated.

In the remainder of this paper, we describe the mechanisms that support dynamic processor

usage modes in Puma. In the next section, we describe the processor modes that are supported from an application perspective. In Section 3, we describe the Puma kernel structures and portions of the kernel interface that support dynamic processor usage modes. In Section 4, we describe the library interface that we have developed to better match the application programmer's model of processor modes. Finally, in Section 5 we describe related work.

2 Processor Modes

In Puma, all of the resources on a node are managed by a single processor, the *system processor*. This is the only processor that performs any significant processing in supervisor mode. The remaining processors run application code and only rarely enter supervisor mode. These processors are called *user processors*. This arrangement produces a slight asymmetry in the performance of the processors and may complicate the application's management of processor cycles. However, it greatly simplifies the structure of the Puma kernel and makes more of the processor cycles available to the applications.

2.1 Heater Mode

The simplest processor usage mode is to run both the kernel and application task on the system processor. We call this arrangement "heater mode" because the user processors only provide heat. Figure 1 illustrates heater mode for two processors.

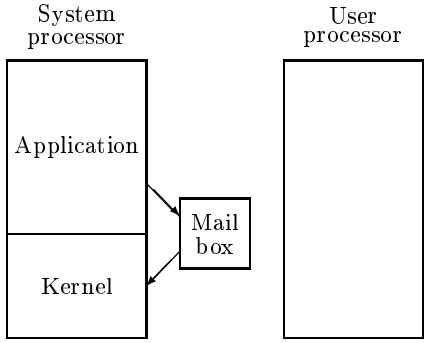

Figure 1: Heater Mode

The "mailbox" shown in Figure 1 is used to pass arguments from the application task to the kernel and back. In Puma, each task has a mailbox. The use of a mailbox instead of the call stack complicates the application's interface to the kernel, but improves overall performance and generalizes to the other processor usage modes easily. The performance improvement stems from the fact that the kernel does not need to validate the stack space used for parameter passing—the mailbox is in a fixed location and known to be at a valid address. The added complexity is easily hidden in the library calls that eventually result in a kernel call.

The mailbox provides the following fields:

- space for the arguments passed to the kernel,
- space for the kernel return value,
- a "mailbox busy" flag,
- a "run kernel" flag, and

- a "kernel done" flag.

When the processors are used in heater mode, each kernel call copies the arguments to the mailbox, sets the "run kernel" flag in the mailbox, and issues a *trap* (software interrupt) instruction to invoke the kernel. When the kernel has finished processing the request, it sets the return status in the mailbox, sets a "kernel done" flag in the mailbox, and re-establishes the context of the task. (The "mailbox busy" and "kernel done" flags are not necessary in this mode; their importance will become apparent when we describe the remaining modes.)

Heater mode does not offer any significant advantages for the application programmer. However, this is the easiest mode to make operational and is, for historical reasons, the default processor mode.

2.2 Kernel-Coprocessor Mode

Figure 2 illustrates kernel-coprocessor mode. In this processor arrangement, the kernel runs on the system processor and the application task runs on the user processor. When the processors are configured in this mode, the kernel polls the external devices and the application mailbox, looking for work.

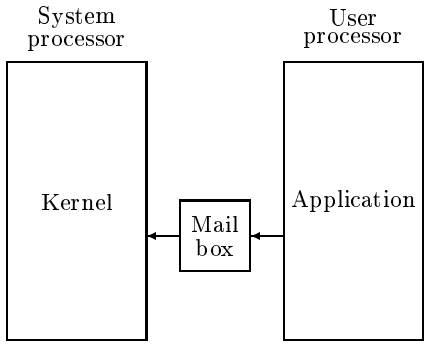

Figure 2: Kernel Coprocessor Mode

When the processors are used in kernel-coprocessor mode, each kernel call clears the "kernel done" flag, copies the arguments to the mailbox, sets the "run kernel" flag in the mailbox, and polls the "kernel done" flag. When the kernel notices that the "run kernel" flag has been set, it will process the request. When it has finished, the kernel sets the return status in the mailbox, and sets the "kernel done" flag in the mailbox.

Because the time to transition from user mode, to supervisor mode, and back to user mode can be very significant, this mode offers the advantage of fast kernel calls. To support this claim, we measured the time taken in a kernel call for Puma running on the Intel i860 processors of an Intel Paragon. When the application and kernel share the same processor (heater mode), a null kernel call[1] takes 5.5 microseconds. When the kernel runs on a separate processor, the same kernel call only takes 3.8 microseconds.

2.3 User-Coprocessor (cop) Mode

Figure 3 illustrates user-coprocessor mode. In this mode, the system processor and user processors both run kernel and application code. However, the kernel code running on the user processor does

[1] The null kernel call does not perform any processing, it simply returns to the application.

not perform any of the resource management activities, it only notifies the system processor when there is a request in the mailbox.

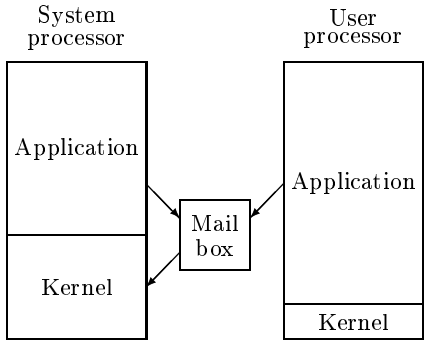

Figure 3: User Coprocessor Mode

When the processors are used in user-coprocessor mode, each kernel call first waits until the "mailbox busy" flag is clear and sets this flag (this can be implemented at user level using a test and set or swap operation). Next, the task clears the "kernel done" flag, copies the arguments to the mailbox, sets the "run kernel" flag in the mailbox, and issues a *trap* instruction. The kernel on the user processor handles the trap by issuing an interrupt to the system processor and returning to the application which then polls the "kernel done" flag. When it has finished handling the request, the kernel on the system processor sets the return status in the mailbox, sets the "kernel done" flag in the mailbox and restores the context of the task running on the system processor. When the requesting task notices that its request has been completed, it copies the return value and clears the "mailbox busy" flag.

The advantage of this mode is that it provides more processor cycles for the application task. Admittedly, the two processors are not symmetric: the part of the task running on the system processor will not progress as rapidly as the portion of the task running on the user processor. This may complicate the application's management of processor cycles; however, a fully symmetric solution would have increased the complexity of the Puma kernel and decreased the total number of cycles available to the application.

2.4 Using More Than Two Processors

The processor usage modes that we have described are based on two processors. If a node has three or more processors, the essential concerns are whether a portion of the task shares the system processor with the kernel and whether there are multiple processors running portions of the task. Having a portion of the task running on the system processor provides additional processor cycles, but requires that applications issue a *trap* instruction to activate the kernel on the system processor. Having multiple processors running portions of the task also provides more processor cycles, but requires that the portions of the task wait while the shared mailbox resource is being used by another processor. (We are looking into the possibility of having a mailbox per task per processor.)

2.5 Dynamic Switching

One of the significant advantages of kernel-coprocessor mode is improved message passing performance. When a portion of the task is running on the system processor, the arrival of a message generates an interrupt which transfers control from the task to the kernel. Similarly, to send a message the task must trap into the kernel. These transfers of control to the kernel and back to the

task on a single processor are a significant portion of the latency associated with message passing. By only running the kernel on the system processor, the application programmer can avoid these transfers and thereby reduce the latency of message passing. As an example, message passing latencies on the Intel Paragon drop from 25 microseconds to 17 microseconds by using the equivalent of kernel-coprocessor mode.

Thus, kernel-coprocessor mode is beneficial for applications that are bounded by message passing latencies. Similarly, user-coprocessor mode is beneficial for applications that are bound by the availability of processor cycles. Many applications transition between phases when they are bound by message passing performance and phases when they are bound by the availability of processor cycles. To support these applications, Puma provides mechanisms that allow the application to switch between the various modes dynamically.

3 Kernel Structures and Interface

The portion of the kernel interface that supports dynamically changing the processor mode is defined by a single kernel call and a small collection of shared data structures. The kernel call is used to allocate or de-allocate a processor. The shared data structures control the portion of the task being run by a processor.

In Puma, the process control block (PCB) for an task is split between the kernel and the application. The two pieces are called the application's PCB and the kernel's PCB. The application's PCB can be read and updated when a processor is in user or supervisor mode. The kernel's PCB can only be read and updated when a processor is in supervisor mode.

Among other things, the application's PCB has a full processor context for each processor. Each processor context includes things like the program counter, stack pointer, condition codes, and so forth. Leaving the processor contexts in a user level data structure makes it possible for the application task to build and manipulate processor contexts without invoking a kernel operation. To protect the integrity of the system, privileged portions of the processor state (e.g., processor mode) are set to safe values when the kernel loads one of these processor contexts onto a processor.

The kernel's PCB for an task includes a set of flags indicating which processors are currently allocated by the task. When an application is scheduled for execution, the dispatch step involves identifying the active application and copying the allocated processor flags to a fixed location in the kernel. The system and user processors poll these flags (the system processor also polls for hardware events), looking for work. Each of the allocated processors uses the active application's PCB to identify and load its context. Any unallocated processors continue to poll the allocation flags.

When the system processor identifies the end of an application's time quantum, it sends each of the user processors a doorbell interrupt. The user processors respond to this interrupt by saving their context in the application's PCB. After they have saved their state, the user processors set a flag to indicate that they are ready for the next application and enter a loop waiting for the system processor to establish the next application. Once all of the user processors have saved their states, the system processor saves and clears the processor allocation flags. When the system processor indicates that it has finished saving the application's state, the user processors return to polling the processor allocation flags, waiting for work.

Any processor can deallocate any other processor. Deallocating a processor that is not currently allocated has no affect. If all processors are deallocated the application is terminated.

When the system processor is deallocated, it simply stays in the kernel code polling for external requests. When a user processor is deallocated, the system processor sets the processor allocation flag to indicate that the user processor is no longer allocated and interrupts the user processor. Upon receiving the interrupt, the user processor notes that it is no longer allocated and simply enters its polling loop.

To run a portion of its code on another processor, a task first builds a processor context in its

PCB. Then it calls the kernel to allocate the processor. This makes it relatively easy for a task to enter user-coprocessor mode.

Entering kernel-coprocessor involves deallocating the system processor. Prior to deallocating the system processor, the application needs to migrate to one of the user processors. This involves copying the current context into the appropriate context buffer, allocating the new user processor, and deallocating the system processor.

4 The Library Level

While there is nothing conceptually difficult in building processor contexts, allocating processors, and deallocating processors, the details change from processor to processor and the code is difficult to debug. To better match the application programmer's expectations, the Puma application library provides the following three routines:

```
int proc_migrate( int processor );

int proc_exec( int processor, void (* code)( void ), void *stack );

int proc_dealloc( in processor );
```

The *proc_migrate* routine is used to migrate a portion of the task from the processor it is currently running on to the processor identified by the argument. If the target processor is currently allocated, *proc_migrate* returns a nonzero result and the application remains on the processor it was using. Otherwise, *proc_migrate* returns zero and the application will be running on the target processor.

The *proc_exec* routine is used to start running a portion of the task on another processor. If the target processor is currently allocated, *proc_exec* returns a nonzero result; otherwise, it returns zero and the application will be running the routine identified by *code* on the target processor.

The *proc_dealloc* routine deallocates a processor. A return value of zero indicates successful deallocation. A nonzero return value indicates that the target processor was not allocated. As should be obvious, this routine does not return if the target processor is the same as the processor on which the routine is executed.

While these routines reflect a higher level interface than the interface provided by the kernel, we recognize that they still represent a fairly low-level interface. To use these routines, the application programmer must still manage stack space for the allocated processors and must keep track of which processors are currently allocated. Here, we expect that a higher level runtime support system, for example Cilk [3], might be used to hide these details from non-systems programmers.

5 Related Work

The NX message passing system [10] under OSF 1/AD on the Intel Paragon uses one CPU as a message coprocessor. The implementation differs from kernel-coprocessor mode in that only the message passing primitives are run on the system processor. All other kernel calls are handled by the user processors. The option to use message coprocessor mode is selected at the time the kernel is compiled. No dynamic reconfiguration is possible.

The PEACE message-passing kernel [6] runs on MANNA nodes. These nodes have two i860 processors sharing a local memory, an I/O interface, and a network connection; much like the nodes of an Intel Paragon. PEACE allows the two processors to be used in one of several modes. The two main modes are message-passing coprocessor mode and asymmetric multiprocessor mode. The former mode corresponds to Puma's kernel-coprocessor mode except that the locking of the mailbox

in shared memory is handled differently. The asymmetric multiprocessor mode of PEACE is similar to Puma's user-coprocessor mode. Which mode is being used is determined when the PEACE kernel is configured.

Remote queues [4] and an active message implementation based on remote queues [7], show that the message coprocessor model has its limitations. User defined handlers cannot be run on the message coprocessor because they might dead-lock the node and eventually the whole system. The active message implementation avoids this problem by compiling often used handlers directly into the kernel code that is executed on the system processor.

Several studies have looked at the cost of context switches [8, 9, 1]. Our kernel-coprocessor mode preserves the caches and TLBs of the individual CPUs, thereby eliminating one of the main costs in a typical context switch.

6 Acknowledgements

Stephen Wheat designed and implemented static versions of these processor modes for SUNMOS (Sandia and University of New Mexico Operating System), the predecessor to Puma. Many members of the Puma team at Sandia and UNM as well as the Lightweight Kernel team at Intel have contributed to the current design and implementation.

We would also like to thank the management at Sandia National Labs who have supported us in our efforts to develop the SUNMOS and Puma operating systems.

References

[1] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In ASPLOS 91 [2], pages 108–120.

[2] *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, CA, April 1991. ACM Press, New York. Published as SIGPLAN Notices, volume 26, number 4.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 207–216, Santa Barbara, CA, July 1995. Published as ACM SIGPLAN Notices, volume 30, number 8.

[4] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Santa Barbara, CA, July 1995.

[5] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[6] Thomas Garnatza, Ute Haack, Michael Sander, and Wolfgang Schröder-Preikschat. Experience made with the design and development of a message-passing kernel for a dual-processor-node parallel computer. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS-29)*, Maui, USA, January 1996.

[7] Lok Tin Liu and David E. Culler. Evaluation of the Intel Paragon on active message communication. In *Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995.

[8] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In ASPLOS 91 [2], pages 75–84.

[9] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, Anaheim, California, June 1990. USENIX Association.

[10] Paul Pierce and Greg Regnier. The Paragon implementation of the NX message passing interface. In *SHPCC 94*, 1994.

[11] Lance Shuler, Rolf Riesen, Chu Jong, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma operating system for massively parallel computers. In *Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995.

[12] Tom Thompson. The world's fastest computers. *Byte*, 21(1):45–64, January 1996.

[13] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. *Scientific Programming*, 3:275–288, 1994.